Guidelines for pipeline interoperability using containers

Authors

Balazs Laurenczy (SIS / ETHZ)
Alexander Kashev (SciTS / UniBe)
Heinz Stockinger (SIB Technologies)
Pablo Escobar López (sciCORE / UniBas)
Sergio Maffioletti (S3IT / UZH)

Purpose of this document

The purpose of this document is to report on the eSCT Pipeline Interoperability project, as well as to provide guidelines to reproduce the important milestones achieved in this project. By following these guidelines, an institution should be able to achieve the same level of interoperability as all the other project members. This document also contains a description of the know-how acquired and some best practices.
Table of content

<table>
<thead>
<tr>
<th>Authors</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose of this document</td>
<td>1</td>
</tr>
<tr>
<td>Table of content</td>
<td>2</td>
</tr>
<tr>
<td>1. Executive summary</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Motivation &amp; Goal</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Main achievements</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Milestones</td>
<td>5</td>
</tr>
<tr>
<td>2. Guidelines</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Singularity</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Installation / configuration of Singularity</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Installing from sources</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Create an RPM or DEB package</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 Singularity configuration</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Building Singularity containers</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Building by bootstrapping (Singularity definition file)</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Building based on Docker</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2.1 Pulling a Docker image directly with Singularity</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2.2 Specify the Docker image during bootstrapping</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2.3 Build from a Dockerfile or a Docker image</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Example pipelines</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1 Basic “hello_world”</td>
<td>11</td>
</tr>
<tr>
<td>2.4.2.1 Building from a Dockerfile with conversion</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1.2 Pulling “hello_world” using Singularity</td>
<td>12</td>
</tr>
<tr>
<td>2.4.1.3 Running the pipeline</td>
<td>12</td>
</tr>
<tr>
<td>2.4.1.4 Runner script and validation</td>
<td>13</td>
</tr>
<tr>
<td>2.4.2 Bioinformatics “pipeline2”</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2.1 Build using a Singularity bootstrap file</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2.2 Pull the container from DockerHub.</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2.3 Running the pipeline</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2.4 Validation of container via MD5 checksum</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3 Additional example “sib-publications”</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3.1 General workflow</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3.2 Building the Docker image from the Dockerfile</td>
<td>17</td>
</tr>
<tr>
<td>2.4.3.3 Converting the Docker image to a Singularity container</td>
<td>17</td>
</tr>
<tr>
<td>2.4.3.4 Running the application</td>
<td>17</td>
</tr>
</tbody>
</table>
3. Best practices & technical topics addressed
   3.1 Path binding 18
   3.2 Container Validation 19
   3.3 Container metadata and runner script 19
   3.4 Container design strategies 21

4 Outlook & open questions 22
   4.1 Container Security 22
   4.2 Advanced HPC and containers 23
   4.3 Shifter and Singularity 23
   4.4 Other interoperability aspects 23

Appendix 25
   A. Singularity bootstrap definition file for bioinformatics pipeline2 25
   B. SLURM submission script for bioinformatics pipeline2 26
   C. Full runner script example for bioinformatics “pipeline2” 28
1. Executive summary

1.1 Motivation & Goal

In the context of the Swiss Personalised Health Network and other precision medicine related initiatives, it will be crucial to be able to run data analysis pipelines in a compatible and reproducible manner across the different HPC clusters of Switzerland. This approach of the “code moving to the data”, rather than moving the data to the computing infrastructures, is motivated by the sensitive character of medical data, as well as the potential large size of data sets. Therefore, a technology that can facilitate the sharing of pipelines with minimal compatibility issues was required.

The aim of this project was to assess whether containerisation could address this need of sharing and running pipelines in a reproducible manner on different HPC clusters. A secondary goal was to create a community around these container technologies in order to facilitate development, deployment and running of these containers.

1.2 Main achievements

In this project, several milestones were defined and completed by all the members. These milestones, described below, were aiming to progressively achieve the goal of this project, starting with the installation of the Singularity container technology, the developments of test pipelines and finally the creation of a container building and testing script made for end-users. In addition to the testing and the gathering of the know-how, this guidelines document is the tangible output of this project, as it can be reused and shared by others.

This project fulfilled its primary goal, by testing and successfully using the Singularity container technology to share pipelines between the HPC clusters of the members. In short, 3 different pipelines with different complexities were built, tested and validated by all the members of the project. Therefore, it is now possible to share containers encapsulating pipelines (or their dependencies) and validate them between the infrastructures of the members of this project.
1.2.1 Milestones

The milestones of this project were defined as follows:

1. **“hello_world” pipeline:** the first milestone consisted in having a basic "hello world" pipeline running in a Singularity container on each infrastructure. This pipeline helped to do a basic validation of the Singularity installation on each site:
   a. Run basic commands from within the container (echo, ls, etc.)
   b. Run some libraries / programs from within the container (python, etc.)
   c. Mount folder from outside the container to be visible inside the container
   d. Read from and write to the mounted folders

2. **Docker to Singularity conversion:** the second milestone consisted in having a workflow for container building based on Docker definition files:
   a. Create some pipelines in Dockerfile format
   b. Create the Docker image and convert it to Singularity container
   c. Alternatively, use the new Singularity version that can directly go from a Docker image to a Singularity container. The Docker image was either in a Dockerfile, on Dockerhub or on a private registry.
   d. run the pipelines with the container created this way (Docker -> Singularity -> run)

3. **Runner script & guideline document:** The final milestone consisted in having a runner script to easily build, test and validate containers, as well as to write guidelines on how to obtain a pipeline interoperability certification:
   a. Runner script
      i. Get the Docker image from Dockerhub, from a Dockerfile or from a private repository (e.g. GitLab)
      ii. Convert the Docker image to a Singularity container, ideally without needing root and / or Docker installed
      iii. Validate the container using a script or command provided by the container's creator (or provider). This validation would eventually use some test input data that should also be available.
   b. Guidelines: the writing of this document.

As said, all members of the project successfully completed the 3 above defined milestones and were able to share, run and validate the test containers used in this project. The following table (Figure 1) summarizes the steps and details of the milestones and sub-milestones of this project.
<table>
<thead>
<tr>
<th>Milestone</th>
<th>UniBe</th>
<th>sciCORE UniBz</th>
<th>$3$I$ T$ UZH</th>
<th>Vital-IT / SIB</th>
<th>SIS ETHZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singularity installed</td>
<td>v2.3</td>
<td></td>
<td>v2.3-dist</td>
<td>(v2.2.1 and v2.3)</td>
<td>v2.3</td>
</tr>
<tr>
<td>&quot;hello world&quot; pipeline building</td>
<td></td>
<td></td>
<td>(from Dockertie)</td>
<td>(from Dockertie)</td>
<td></td>
</tr>
<tr>
<td>&quot;hello world&quot; pipeline running</td>
<td></td>
<td></td>
<td>(from Singularity image)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sciCORE pipeline1 &amp; pipeline2 building</td>
<td></td>
<td></td>
<td>(only pipeline 2. pipeline1 is too big to run in home directory (quota issue))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sciCORE pipeline1 &amp; pipeline2 running</td>
<td></td>
<td></td>
<td>(with Singularity image)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Docker -&gt; Singularity conversion via Dockertie</td>
<td>(with &quot;hello world&quot; pipeline)</td>
<td></td>
<td>(with SIB-publications pipeline)</td>
<td></td>
<td>(with &quot;hello world&quot; pipeline)</td>
</tr>
<tr>
<td>Docker -&gt; Singularity conversion via Dockerhub</td>
<td>(with &quot;whalesay&quot; container)</td>
<td>(with custom Dockerties for pipeline1 &amp; pipeline2)</td>
<td>postracklab/mxiprep</td>
<td>not planned</td>
<td>(with &quot;whalesay&quot; container and custom Dockerties)</td>
</tr>
<tr>
<td>Docker -&gt; Singularity conversion via &quot;private&quot; Gitlab</td>
<td>(with &quot;hello world&quot; pipeline, no auth)</td>
<td></td>
<td>not planned</td>
<td>(from private GitLab)</td>
<td></td>
</tr>
<tr>
<td>Runner script</td>
<td>(pipeline.py)</td>
<td>(pipeline.py from UniBe)</td>
<td>(own runner script, tested with SIB-publications pipeline)</td>
<td>(pipeline.py from UniBe)</td>
<td></td>
</tr>
<tr>
<td>Guideline document</td>
<td></td>
<td></td>
<td>all details are in: README.md and in final report</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.** The milestone completion table, summarising the progress of each institution regarding each milestone, including the details of how the milestones were completed.

In addition, this project led to the creation of a very interactive **community** of (future) container developers, deployers, maintainers (and possibly runners), facilitating the future exchange of pipelines and the resolution of related problems.
2. Guidelines

2.1 Singularity

In this project, we used Singularity (http://singularity.lbl.gov/), which is an HPC friendly container technology. Singularity also has a high degree of compatibility with Docker (one of the most mature container technology). As explained later, Docker containers are used as the main base for exchanging containers, due to the widespread usage and availability of Docker containers. Singularity is mainly used as a way of running Docker containers on HPC environment.

A simple Singularity call looks as follows, given that Singularity is installed and available in the PATH of the user:

$ singularity run myImage.img myCommandToRunInContainer

In the following sections, all the prerequisites for being able to run such a command will be described, including installation of Singularity, building of the container using various methods, and we then provide some example test pipelines using Singularity containers.

2.2 Installation / configuration of Singularity

2.2.1 Installing from sources

Installation from source is pretty straightforward, you only need to run:

- ./configure
- make
- sudo make install

There are no special dependencies besides a compiler. sudo permissions are only required for the latest make install step as Singularity needs to set the setuid bit for some binaries. Please refer to the official documentation for details:

http://singularity.lbl.gov/install-linux#option-1-download-latest-stable-release

2.2.2 Create an RPM or DEB package

If preferred, you can create a package (rpm or deb) for your distribution. Please refer to the official documentation for details:

http://singularity.lbl.gov/install-linux#build-an-rpm-from-source
http://singularity.lbl.gov/install-linux#build-a-deb-from-source
2.2.3 Singularity configuration

Singularity provides a **configuration file** located in:

- `$PREFIX/singularity/singularity.conf`

where the system administrator can define some settings depending on the site policy or user requirements. Not all the details about available configuration options are described here, but the following are the **most important ones**:

- **mount home = yes** Defines if the user home folder should be available inside the container by default
- **mount tmp = yes** Defines if the `/tmp` folder from the host should be available inside the container by default
- **bind path** Define multiple paths from the host system which should be available inside the container by default. A typical use case is to configure the cluster’s shared scratch file system or the cluster’s projects file system to be available inside the containers.
- **enable overlay = yes** This option is very interesting as it enables the use of the overlay file system (`overlayfs`) so that paths which do not exists inside the container are created on the fly. For example, if your cluster’s home folder are located in `/cluster/parallelsfs/homes` and you do not use `overlayfs`, the path can only be used if it also exists inside all of your containers. By using `overlayfs`, this path is automatically created when the container boots. This options depends on the `overlayfs` kernel module which is only available in recent kernels. We have tested it with the kernel 3.10.0-514.10.2.el7.x86_64 from Centos7.3. With older kernels, some testing is recommended before deploying it in production.

For further details about all the available configuration options please refer to the official documentation: [http://singularity.lbl.gov/docs-config](http://singularity.lbl.gov/docs-config)
2.3 Building Singularity containers

In order to run a Singularity application, one first needs to have a Singularity container, which basically consists of a single container file (see http://singularity.lbl.gov/quickstart). In the project, we experimented with several different methods to build Singularity containers.

2.3.1 Building by bootstrapping (Singularity definition file)

The Singularity definition file (bootstrap file) contains all information to install a basic operating system and all necessary packages for running the final application. One can either use Docker commands (see below) or yum (on RedHat and CentOS-based systems) to install packages for an image.

This method is however not compatible with Docker as the Singularity definition file can only be used to build Singularity containers. As those containers cannot be reverted to Docker, they cannot take advantage or contribute to the existing Docker ecosystem and containers. Therefore, we do not go into further details here and we recommend using the building method described later that is Docker compatible (see 2.3.2 Building based on Docker). Please consult the official documentation about how to build a container using the Singularity definition file (http://singularity.lbl.gov/bootstrap-image).

Note: using a Singularity definition file currently requires root access (might change for future Singularity versions), regardless of the bootstrap method used (Docker, yum etc.).

2.3.2 Building based on Docker

Using a Dockerfile (or an existing Docker image) as the starting point is the preferred and recommended way to obtain Singularity images since Docker is already quite known and widespread in the community. This allows to share containers on the level of Docker, while keeping it compatible with HPC environment by converting them to Singularity.

There are several methods to build a Singularity container from a Docker image or a Dockerfile.

2.3.2.1 Pulling a Docker image directly with Singularity

If a Docker image is already uploaded to Dockerhub or another Docker registry, one can use the singularity pull command. This command takes care of fetching the Docker image and convert it to a Singularity image file. Importantly, this does not require root:

$ singularity pull docker://docker/whalesay:latest

Useful options are:

- --size 2048 - container size in MB, may be required for some images.
- --name pipeline.img - overrides the auto-generated container name.
This command may be used with private Docker registries as well. In general, the syntax for the URI is:

```
docker://<registry>/<namespace>/<repo_name>:<repo_tag>
```

If needed, please see the documentation on how to pass credentials for the registry (http://singularity.lbl.gov/docs-docker#custom-authentication).

### 2.3.2.2 Specify the Docker image during bootstrapping

It is possible to use a Docker registry URI as a source for bootstrapping in a Singularity file (see previous section on the Singularity definition file). Note that this requires root access.

**Bootstrap:**
```
  From: tensorflow/tensorflow:latest
```

### 2.3.2.3 Build from a Dockerfile or a Docker image

If you have a Dockerfile instead of a ready Docker image, you will need to have **Docker installed** to build the Docker image. This requires root access, which means that this is typically not done on the HPC infrastructure, but rather on a local laptop. For more information about how to build Docker images from a Dockerfile, please refer to the official Docker documentation (https://docs.docker.com/engine/reference/builder/).

Once the Docker image is available, it can be converted to a Singularity image using **docker2singularity** (https://github.com/singularityware/docker2singularity), which itself is a Docker container provided by the Singularity developers to convert Docker images to Singularity images, very similarly to what happens during the `singularity pull` command.

To convert a Docker image into a Singularity container, one can use the following command:

```
$ docker run \
- v /var/run/docker.sock:/var/run/docker.sock \
- v $(pwd)/output \
--privileged -t \ 
--rm singularityware/docker2singularity \
myDockerImageName:myTag
```

This **conversion also requires a working Docker installation**, as well as the existence of the Docker image (`myDockerImageName:myTag` in the example above). This step, as well as the Docker image building step are therefore **typically executed outside of the HPC environment**. This conversion will create a Singularity image in the current working directory (`$(pwd)`). The output directory can be modified by specifying a path instead of the `$(pwd)` command. For more details, please see the example pipelines section below. Note that there are compatibility guidelines to keep in mind when building a Dockerfile for use with
Singularity (see https://github.com/singularityware/docker2singularity#tips-for-making-docker-images-compatible-with-singularity).

2.4 Example pipelines

In order to get practical experience with building and running containers, several different pipelines were created and tested:

1. basic “hello_world” (provided by ETH/SIS)
2. bioinformatics “pipeline2” (provided by UniBas/sciCORE)
3. Additional example “sib-publications” (provided by SIB)

Source code and instructions are available in Git in the following repository in the “public” branch:

https://c4science.ch/diffusion/2915/browse/public/

and can be cloned as follows (use the “public” branch):

git clone -b public https://c4science.ch/diffusion/2915/esct-pipeline-interoperability.git

Here, we outline these examples and how they can be used. The different pipelines described below use different strategies for wrapping the pipelines into the containers, either by full wrapping (basic “hello_world”), wrapping of dependencies (bioinformatics “pipeline2”) or by wrapping the environment while still exposing the execution script (“sib-publications”). These different strategies are further discussed below in the “Best practices” section (see 3.4 Container design strategies).

2.4.1 Basic “hello_world”

The “hello_world” pipeline is a very basic test pipeline to assess whether your Singularity installation is working and whether you can run a simple pipeline. The container contains a basic “hello world” Snakemake pipeline that simply writes strings to files using echo (for more information about the Snakemake workflow manager, please see http://snakemake.readthedocs.io). This pipeline is intended to be tested either on local virtual machine or on the HPC machines directly. For more information about virtual machines for testing Singularity (e.g. Vagrant), please see the Singularity documentation for setting up a test environment at http://singularity.lbl.gov/install-mac#option-1-singularityware-vagrant-box.

2.4.2.1 Building from a Dockerfile with conversion

To build the “hello_world” pipeline’s container, you can use the Dockerfile available at:

https://c4science.ch/diffusion/2915/browse/public/example_pipelines/hello_world/Dockerfile
With that file, you can **build the Docker image and convert it to a Singularity container** following the procedure described above (see [2.3.2.2 Build from a Dockerfile or a Docker image](#)).

Alternatively, a short bash script is also provided to do these steps in one call:

```bash
#!/usr/bin/env bash

# Author: Balazs Laurenczy (ETHZ)
# Date: 2017-06-30
# This script requires Docker and Singularity to be installed.
# Run this in a vagrant virtual machine or alike.

# remove old container(s)
rm -rf *.img

# build the Docker image using the "Dockerfile"
sudo docker build -t hello_world_pipeline .

# convert the Docker image to a Singularity image
sudo docker run -v /var/run/docker.sock:/var/run/docker.sock -v $(pwd):/output
--privileged -t --rm singularityware/docker2singularity hello_world_pipeline

# rename the image
mv hello_world_pipeline*.img hello_world_pipeline.img
```

(source: [https://c4science.ch/diffusion/2915/browse/public/example_pipelines/hello_world/create_and_run_container_locally.sh](https://c4science.ch/diffusion/2915/browse/public/example_pipelines/hello_world/create_and_run_container_locally.sh))

### 2.4.1.2 Pulling “hello_world” using Singularity

The container can also easily be **fetched from Dockerhub** as described above ([2.3.2.1 Pulling a Docker image directly with Singularity](#)) using the following `pull` command:

```
$ singularity pull docker://blaurenczy/esct-pipeline-interoperability:hello_world
```

Alternatively, the container can also be **downloaded from a “private” Docker registry** in a similar way as you can download it from the public Dockerhub registry. Here is the command to download the image from the SIS (ETHZ) GitLab repository:

```
$ singularity pull --size 1024
docker://sissource.ethz.ch:5005/balazsl/esct-pipeline-interoperability:hello_world
```

### 2.4.1.3 Running the pipeline

To run the “hello_world” pipeline, you need to have the following files / folders in your current working directory:

- **./data** - folder where the output data will be written. This folder will be mounted to the container under the path `/data`. This folder also contains the reference output (`ref_output`) that will be used to validate the container and the pipeline.
- **./pipeline** - folder where the pipeline’s logic is (Snakemake file). This folder will also be mounted to the container under the path `/pipeline`. 
The pipeline can be then executing by running the container as follows:

```bash
$ singularity run -B ./pipeline:/pipeline -B ./data:/data hello_world_pipeline.img
```

This command invokes the `singularity executable with the “run” instruction`, meaning that the `entry point` defined inside the container will be executed. This also will bind the `.pipeline` and the `.data` folders from the host to folders in the container. Please see more details about this below (3.1 Path binding).

The entry point is defined in the Dockerfile used to build this container and is the following: (available at https://c4science.ch/diffusion/2915/browse/public/example_pipelines/hello_world/Dockerfile)

```bash
ENTRYPOINT ["snakemake", "-q", "-s", "/pipeline/hello_world_pipeline.snake", "-d", "/data"]
```

which means that the container will execute the following upon being called via `run` to:

```bash
$ snakemake -q -s /pipeline/hello_world_pipeline.snake -d /data
```

This means that the `snakemake` executable, installed inside the container, will call in quiet mode (-q) the snakemake pipeline script (-s) located at `/pipeline/hello_world_pipeline.snake` using the working in the `/data` directory. Note that both `/data` and `/pipeline` are folders existing inside the container only, as they were mounted from the host to the container.

The output of the run should be identical (or very similar) to the following:

```
$ singularity run -B ./pipeline:/pipeline -B ./data:/data hello_world_pipeline.img
#2017-06-30 15:17:39,892 | INFO    | hello_world_pipeline.snake: Starting Snakemake pipeline ...
#2017-06-30 15:17:40,004 | INFO    | hello_world_pipeline.snake: Starting Snakemake pipeline ...
#2017-06-30 15:17:40,034 | INFO    | hello_world_pipeline.snake: Pipeline is valid.
```

If you see this, then it means that the Snakemake pipeline inside the container successfully run, successfully created the output files it was supposed to and successfully validated these output files by comparing the MD5 checksum against the reference output. Note that self-validation is not the best practice regarding container validation, as discussed later (see 3.2 Container Validation).

2.4.1.4 Runner script and validation

Ultimately, the easiest and most robust way of building, testing and validating this container is by using the runner script as described later (see 3.3 Container metadata and “runner” scripts).
Briefly, here are the commands and the expected output to **build** the “hello_world” pipeline using the runner script:

```bash
$ python3 pipeline.py build -p hello_world.yaml
Loading pipeline description...
Pipeline 'hello_world' 1 loaded.
Target image file: hello_world-1.img

Building pipeline...

Executing step 1: singularity pull --size 512 --name hello_world-1.img
docker://sissource.ethz.ch:5005/balazsl/esct-pipeline-interoperability:hello_world

Initializing Singularity image subsystem
Opening image file: hello_world-1.img
Creating 512MiB image
Creating file system within image
Image is done: hello_world-1.img
Docker image path: sissource.ethz.ch:5005/balazsl/esct-pipeline-interoperability:hello_world
Cache folder set to /home/vagrant/.singularity/docker
Importing: base Singularity environment
Importing: /home/vagrant/.singularity/docker/sha256:b6f892c0e03b3b8134ea421d706f8ee6216cabcbe7e6e3a4527ed379f921b.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:55b69f332b04767e792a3619fae0491852c1440c2da6ecb3452cc88efcfc1f6b.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:3a2ef3f3b0e7245571b0d1292de65f2b0731b099e992bec662d1e01979535d.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:c7f722e606adad11895f080ba542a181c91a19606660b15155991341930e.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:9a909602e9c94cf7e29f9f0f035c32c67a9a0085108c5350ba47f146c7df.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:86556787dbd5e8e3b2d329280bd743e3787470f8afe1e2c6dd063d3d4b9f0e.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:f5ee237ca2ac0718090488e609293bbc3607a50b397a1c26a8456e5e7f1f2c.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:68d97737ca3c6e3f3a0d886fe890d9c36f04f65fa68f646a05a2e2f7563.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:4a3a45429e02db26d6d7c3a6f164a785fa945552486fc3b0eda3e38481d8de.tar.gz
Importing: /home/vagrant/.singularity/docker/sha256:7f9d25ad2f1b9c28dfc49ace2e76dbf850494f118c37372a2c3e7196786ff5f.tar.gz
Importing: /home/vagrant/.singularity/metadata/sha256:3609f33c2cedb0a3773aa565b10f25a6d46c9ca31e0d10975974946811d6f95.tar.gz
Done. Container is at: hello_world-1.img
Successfully built image hello_world-1.img.
```

And here are the commands and the expected output to **test and validate** the “hello_world” pipeline using the runner script:

```bash
$ python3 pipeline.py test -p hello_world.yaml
Loading pipeline description...
Pipeline 'hello_world' 1 loaded.
Target image file: hello_world-1.img

Testing pipeline...

(Re)creating test files...


```
Resolving polybox.ethz.ch (polybox.ethz.ch)... 129.132.71.228, 129.132.71.230, 129.132.71.229, ...
Connecting to polybox.ethz.ch (polybox.ethz.ch)|129.132.71.228|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1715 (1.7K) [application/x-tar]
Saving to: 'hello_world_input.tar'

100%[================================================================================================>] 1,715 --.-K/s in 0.009s

2017-06-30 15:25:41 (183 KB/s) - 'hello_world_input.tar' saved [1715/1715]
```

Executing step 2: tar xf hello_world_input.tar

Running pipeline...
Executing step 1: singularity run -B ./data:/data -B ./pipeline:/pipeline hello_world-1.img > hello_world.out 2> hello_world.err
Successfully ran hello_world.
Running validation stage...
Executing step 1: diff data/output/content.md5 data/ref_output/content.md5
Executing step 2: cd data/output && find . -regex ".*.log" -type f -exec md5sum {} \; | sort -k2 > ../../content.md5 && diff ../../content.md5 ../ref_output/content.md5
Pipeline validated successfully!

This validation is done outside the container, meaning it is independent from the container’s building, which is necessary for a safe validation (see 3.2 Container Validation).

2.4.2 Bioinformatics “pipeline2”

2.4.2.1 Build using a Singularity bootstrap file

First, create the Singularity bootstrap file named “Singularity” with the content shown in the Singularity bootstrap definition file for bioinformatics pipeline2 (see appendix).

Create an empty container definition file:

```bash
$ sudo singularity create -s 2048 pipeline2.img
```

Then, you can bootstrap (= build) the container to the the empty disk image:

```bash
$ sudo singularity bootstrap pipeline2.img Singularity
```

2.4.2.2 Pull the container from DockerHub.

This step uses direct pulling of the Docker image from Dockerhub. First, create a docker container using a Dockerfile and upload it to Dockerhub.

Here you can see the used Dockerfile:
https://hub.docker.com/r/pescobar/scicore-pipeline2/~/dockerfile/

Here you can see how to upload it to Dockerhub:
https://docs.docker.com/docker-hub/builds/

Then, you can download the container from Dockerhub:

```bash
$ singularity pull -n "pipeline2.img" docker://pescobar/scicore-pipeline2:latest
```

2.4.2.3 Running the pipeline

This is an example submission script to run the pipeline in a SLURM cluster. The steps to follow are:
$ singularity pull -n "pipeline2.img" docker://pescobar/scicore-pipeline2:latest
$ wget http://users.scicore.unibas.ch/~escobar/data/scicore-pipeline2-input-data.tar.bz2
$ tar xf scicore-pipeline2-input-data.tar.bz2
$ sbatch slurm-submit-script-WITH-containers.sh

An example submission script that can be used to submit the job to a **SLURM scheduler** can be found in the appendix (see [bioinformatics pipeline2 SLURM submission script](#)).

An additional submission script for the **LSF scheduler** can be found at [https://c4science.ch/diffusion/2915/browse/public/example_pipelines/bioinformatics_pipeline2/LSF_submit-script-with-containers.sh](https://c4science.ch/diffusion/2915/browse/public/example_pipelines/bioinformatics_pipeline2/LSF_submit-script-with-containers.sh).

### 2.4.2.4 Validation of container via MD5 checksum

This pipeline can be **validated** following these steps:

```bash
$ singularity exec pipeline2.img vcftools --vcf
  output/freebayes/yeast_reseq_ds-bt2aln.s.vcf --FILTER-summary --stdout | md5sum
33921b7c4b3d3f3da638d9aa517c0a07 -
```

### 2.4.3 Additional example “sib-publications”

The main motivation of this pipeline was to develop a **real case scenario** and an application that will be used even after this eSCT pipeline interoperability project. Therefore, the SIB-publications project was chosen. Indeed, this project aims to obtain all publications of SIB members registered in Europe PMC ([http://europepmc.org](http://europepmc.org)) on a monthly basis or manually by specifying the requested year and month of interest. This pipeline was however **not included in the milestones and was not tested by all institutions**. It is provided as an additional example that the reader can test.

Additional motivations were:

- to run a Python program with specific requirement: ICU library for utf-8 sorting
- to automate the application: run it once a month on a compute node of a cluster (originally, this application was run on a personal computer with local system dependencies)
- to easily retrieve the output data
- to allow for small program modifications if necessary
- to hide software complexity

Availability of code: [https://c4science.ch/diffusion/2915/browse/public/example_pipelines/SIB-publications/](https://c4science.ch/diffusion/2915/browse/public/example_pipelines/SIB-publications/).
2.4.3.1 General workflow

This example has a slightly different design as part of the pipeline's code is a python script living outside of the container. We used this design as it has proven very flexible for the specific needs of this project. Indeed, the Docker image is built once and for all with all the necessary dependencies, while the Python scripts can be changed easily, without the need to re-create a new container.

2.4.3.2 Building the Docker image from the Dockerfile

The Docker image consists of CentOS 7, python3.6, system ICU and python pyICU libraries necessary for string sorting, as well as a wrapper.sh shell script to ensure that the LANG ENV variable is exported correctly for Python to work.

We significantly adapted our Dockerfile to ensure compatibility with Singularity and the singularityware/docker2singularity conversion tool:

- usage of root's .bashrc to set up some environmental variables, such as LANG (needed for Python in some cases): we used ENV instructions or equivalent
- usage of CMD: we used ENTRYPOINT instead
- usage of elevated user permissions: we used mounted/binded volume data with appropriate permissions (chmod 755) for input/output
- encapsulating the input data in the container (see previous point about permissions): we put the input data outside the container

Once the above have been modified, we could still successfully run the Docker image.

2.4.3.3 Converting the Docker image to a Singularity container

Using the modified Dockerfile, we successfully converted the Docker image to a Singularity container using the following command:
2.4.3.4 Running the application

In order to **run the application**, one needs the following files/directories:

```bash
$ sib-publications.img         (built with previous step)
$ data/SIB-monthly-publications.py (available in Git)
```

In order to list all SIB publication of the January 2017 (2017 01) one can use the following command to launch the pipeline:

```bash
$ singularity run -B ./data:/data sib-publications.img
    /data/SIB-monthly-publications.py 2017 01
```

All tests were performed on Mac OS X, CentOS 7 (Docker) and the Vital-IT HPC infrastructure (Singularity 2.3.1) using CentOS 6. Detailed instructions for all steps discussed here can be found in the README.md file (see [https://c4science.ch/diffusion/2915/browse/public/example_pipelines/SIB-publications/README.md](https://c4science.ch/diffusion/2915/browse/public/example_pipelines/SIB-publications/README.md)).

3. Best practices & technical topics addressed

3.1 Path binding

The file system present in the container replaces the host’s filesystem for applications running in it; furthermore, this filesystem is by default read-only. To access the data on the host system, certain **parts of the host’s filesystem are bound inside the container**.

A binding can be:

- Inferred automatically (for example the user’s home folder is usually bound)
- Specified globally by Singularity configuration:

  ```
  # /etc/singularity/singularity.conf
  bind path /path/on/host:/path/in/container
  bind path /identical/path/on/both
  ```
- Specified by the user at runtime through flags if allowed by Singularity configuration:
$ singularity run -B ./data:/data singularity_container.img [...]

See singularity help run for more runtime bind options.

Note that Singularity’s access to the bound filesystem parts are defined by the user rights of the account that launched the container.

Normally, for the binding to occur a corresponding path must exist in the image’s filesystem. Depending on the global configuration and host capabilities, Singularity can use “overlay” binding, that partially bypasses the problem above by allowing arbitrary new paths inside the container.

However, the pipeline itself may expect exact locations for common things like scratch space. In that case, binding paths should be documented in the pipeline’s metadata so that the binding can be adapted for particular deployments.

Singularity recommends developing a standard for “bind points” across Singularity images (see http://singularity.lbl.gov/docs-mount). That way, pipelines can expect some uniform locations like /scratch inside the container being populated on the level of a global configuration file for a particular HPC deployment.

### 3.2 Container Validation

As a general rule, any pipeline should provide some easy means of testing that it works in a particular environment. Such testing scenarios should be provided by the pipeline authors alongside other pipeline metadata. It is recommended if those tests are carried outside of the container, as the goal is to establish trust in the container’s output. Therefore, we do not recommend to rely solely on a self-validation procedure included in the containers.

This could consist of well-known sets of data for all components of the pipeline, together with a script to check that the results are as expected. In case of deterministic output, the easiest approach to checking the results is an output of md5sum that can be validated against the results:

```
# In known good environment, prepare well-known-input-data
$ singularity run pipeline.img some-tool well-known-input-data
$ md5sum [...list of output files..] > check.md5
# Record check.md5 and a way to generate well-known-input-data

# In tested environment, regenerate well-known-input-data
$ singularity run pipeline.img some-tool well-known-input-data
$ md5sum -c check.md5
```

In case of non-deterministic output, some more complex verification method should be provided to ensure that it is “sane”.

3.3 Container metadata and runner script

One possible way to collect all the metadata on how to build, run and test a container and standardize the way pipelines are deployed is by using a common runner script combined with a configuration format.

During the course of this project, a prototype Python script was developed. It allows to specify the container build procedure, the steps for running the pipeline and generation/validation of test data.

Up to date sources available at:
https://c4science.ch/diffusion/2915/browse/public/singularity-pipeline-runner/

Example configuration script in YAML format (full example for bioinformatics_pipeline2 in the appendix):

```
name: sciCORE Pipeline 2
version: 0.1

build:
  type: bootstrap  # Supports bootstrap, pull and completely custom commands
  source: Pipeline2 # A Singularity bootstrap file provided together with the configuration
  size: 2048

substitutions: # Strings that can be used in commands
  RefGenome: "./scicore-pipeline2-input-data/some/long/path/

run:
  commands: # An array with a sequence of commands that constitute the pipeline
    - "[ -d output/bowtie2 ] || mkdir -p output/bowtie2" # Preparing output folders
    # Commands can use substitutions like {image} or shell substitutions like ${NUM_CPUS}
    - "singularity exec {image} bowtie2 -p ${{NUM_CPUS}} -x $(RefGenome)/Bowtie2Index/genome [...]" # ...

test:
  test_files: # A list of input files that must exist for the test
    - scicore-pipeline2-input-data.tar.gz
    - scicore-pipeline2-input-data
    - output.md5
  prepare_commands: # A list of commands to generate above files, if they don't exist
    - "wget -O scicore-pipeline2-input-data.tar.gz http://users.scicore.unibas.ch/ [...]"
    # ...
  Validate_commands: # A list of commands to execute after the pipeline is run
    - "singularity exec {image} vcf tools [...] | md5sum -c output.md5"

Usage:

- $ pipeline build
  Will build the image according to build type and source provided. In this example, will use a Singularity bootstrap file.
$ pipeline run
Will check that image exists, and if it does - runs the supplied sequence of commands. Commands can be arbitrary, and use some substituted variables like the image name or user-defined ones.
If any of the commands returns a non-zero status, stops and returns non-zero status.

$ pipeline test
Will check that specified test files exist, otherwise runs the sequence of prepare commands to generate (e.g. download) them.
Then, runs the pipeline as normal.
Then, runs the sequence of validation commands (e.g. checking MD5 hashes of output).

Note that the script does not interface with workload managers; its calls should be wrapped in appropriate batch scripts for your specific manager.

With additional polish, the runner script may be uploaded to PyPi and be available for standard installation through `pip install`.

3.4 Container design strategies
As said above, there are different ways to design containers when their purpose is to encapsulate pipelines. A pipeline is a chain of data-processing software (e.g. bioinformatics workflow) and therefore needs to make successive calls to usually different softwares. These successive calls are usually defined either in a bash script, a python script, or, in case of more complicated pipelines, in a workflow manager language (Snakemake, NextFlow, etc.).

This scripts defined the orchestration of the calls.

It is possible to make the orchestration either inside the container, or to make it from outside the container and simply make the calls to software located inside the container. Figures 3a, 3b and 3c illustrates these differences.

![Figure 3a](image)
Figure 3a. A 3-step pipeline is using two containers that have dependencies installed inside. The pipeline is run on the host via a bash script. Each step is calling a tool located in one of the two containers using the "singularity exec" command.
Figure 3b. A 2-step pipeline is encapsulated in a container. The **ENTRYPOINT** of the container defines the steps to be executed when the container is called using "**singularity run**".

Figure 3c. A 3-step pipeline is encapsulated in a container. A script defining the execution steps (**python** / **bash** / **snakemake** / ...) is called from outside using the "**singularity exec**" command. If the script is mounted inside the container, it can be easily changed from outside without recreating the container.

An orchestration inside the container usually gives simpler containers that just need to be run, and are therefore easy to share and use.

Orchestration from outside the container allows however to easily swap containers, for example in case a new version for a tool inside the container is made available. In addition, orchestration from outside allows to use the HPC cluster’s scheduler to parallelize the steps inside the container, which is impossible in the case of orchestration from inside as the container cannot access the scheduler.

### 4 Outlook & open questions

#### 4.1 Container Security

Containers provide an isolated environment for code to run in, but unlike a virtual machine the containers still interact with the host OS. Additionally, starting a container currently needs some code with elevated privileges.

Singularity uses a more isolated approach to elevated privileges than Docker, which makes it preferable for HPC environments (see [http://singularity.lbl.gov/docs-security#how-do-other-container-solutions-do-it](http://singularity.lbl.gov/docs-security#how-do-other-container-solutions-do-it)). Once the environment is set up for execution of the container, all elevated privileges are dropped and the code running inside the container executes with the same rights as the calling user, and processes within the container cannot elevate their privileges.

Still, Singularity deployments need to follow standard practices to protect against privilege escalation from regular user accounts (especially the kernel, as it is shared with the container) to stop potential malware in containers from running.
For audit purposes, **all pipelines should include the exact steps to produce the image**, preferably contained in form of a Dockerfile / Singularity bootstrap file. The ability to download a prebuilt container from a registry is convenient, but reproducible builds are important, among other things, for security purposes.

Since building a container from scratch requires full root privileges, the standard procedure for developing containers is to build them on a local machine and then move them to the HPC system for execution (see Figure 4).

![Diagram showing the workflow of creating a new container](image)

**Figure 4.** The typical workflow of creation of a new container involves 2 parts: the creation, building and testing of the container locally (left) followed by the usage on the HPC environment (right).

### 4.2 Advanced HPC and containers

Singularity supports integration with some *advanced* HPC features that go beyond the single or multi-CPU use of a single compute node.

Singularity supports the usage of **MPI**, a communication standard that “addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process” ([http://mpi-forum.org/](http://mpi-forum.org/)). Applications relying on MPI can be containerized, shared and run on HPC clusters in a similar way as standard single or multi-core applications (see [http://singularity.lbl.gov/docs-hpc#integration-with-mpi](http://singularity.lbl.gov/docs-hpc#integration-with-mpi)).

Similarly, Singularity also supports the use of **GPUs** within containers (see [http://singularity.lbl.gov/tutorial-gpu-drivers-open-mpi-mlts](http://singularity.lbl.gov/tutorial-gpu-drivers-open-mpi-mlts)).

However, the **current project did not explore these aspects** of the Singularity technology and can therefore not provide guidelines on how to build and use MPI-enabled or GPU-compatible containers. Nevertheless, such technology is definitely useful and worthy of exploring and was **only left out for time reasons**. Among others, usage, compatibility requirements, comparison with native (= non-containerized) performance as well as with other container technologies are all topics of high interest.
4.3 Shifter and Singularity

Singularity is not the only container technology that can work on HPC environment. Shifter is another container technology that “allows an HPC system to efficiently and safely allow end-users to run a docker image” ([https://github.com/NERSC/shifter](https://github.com/NERSC/shifter)). Shifter is well integrated with Docker and can directly pull images from Dockerhub.

The current project did not assess the differences between Singularity and Shifter. However, keeping Docker as the basic sharing technology for containers allows us in the future to easily switch between Singularity and Shifter to deploy and run the containers on the HPC infrastructure.

4.4 Other interoperability aspects

This project was focused to make the different HPC sites compatible regarding containers. However, other interoperability aspects still remain to be solved. Among others, it would be interesting to explore the Common Workflow Language (CWL) to define standardized pipelines, which can thereby be more easily shared irrespective of the scheduler or the workflow manager.

In addition, efforts from other organisations to work on pipeline interoperability should also be examined as they might provide useful ideas and knowledge on how to best leverage containers for pipeline interoperability. Additionally, collaborations with organisms working internationally or in other countries might enable to have pipeline interoperability on a larger scale, going beyond the limits of Switzerland. For example, the container and workflow group of the Global Alliance for Genomic and Health (GA4GH) is “focused on technologies and best practice methodologies for running portable and reproducible genomic analysis pipelines. This group coordinates efforts around the development of methodologies for running repeatable genomic workflows, as well as tools and APIs for discovering genomic analysis tools” ([https://genomicsandhealth.org/working-groups/our-work/containers-and-workflows](https://genomicsandhealth.org/working-groups/our-work/containers-and-workflows)).
Appendix

A. Singularity bootstrap definition file for bioinformatics pipeline

Bootstrap: docker
From: centos:7.3.1611

%runscript
   #"I can put here whatever I want to happen by default when the user runs the container"
   cat << EOF
This container includes the following apps:
Bowtie2/2.2.9 - http://bowtie-bio.sourceforge.net/bowtie2/index.shtml
SAMtools/1.3.1 - http://www.htslib.org/
freebayes/1.1.0 - https://github.com/ekg/freebayes
vcftools/0.1.14 - https://vcftools.github.io
Optional tools only used to download the public data and downsample it:
Seqtk/1.2 - https://github.com/lh3/seqtk
To execute a binary inside the container do "singularity exec /path/to/container.img binary-name"
EOF

%post

   echo "Here we are installing software and other dependencies for the container!"
   echo "Installing bowtie2-2.2.9"
yum -y install unzip wget
   cd /usr/local/src/
   [ -f bowtie2-2.2.9-linux-x86_64.zip ] || wget https://sourceforge.net/projects/bowtie-bio/files/bowtie2/2.2.9/bowtie2-2.2.9-linux-x86_64.zip
   unzip bowtie2-2.2.9-linux-x86_64.zip
   cp bowtie2-2.2.9/bowtie2* /usr/local/bin/

   echo "Installing samtools-1.3.1"
yum -y install make gcc gcc-c++ zlib-devel bzip2-devel ncurses-devel bzip2
   cd /usr/local/src/
   [ -f samtools-1.3.1.tar.bz2 ] || wget https://github.com/samtools/samtools/releases/download/1.3.1/samtools-1.3.1.tar.bz2
   tar xf samtools-1.3.1.tar.bz2
   cd /usr/local/src/samtools-1.3.1
   ./configure
   make
   make install

   echo "Installing freebayes-1.1.0"
yum -y install git cmake
   cd /usr/local/src/
   git clone --recursive git://github.com/ekg/freebayes.git

   # the release system for freebayes does git --recursive which makes hard to reproduce the same installation
   # To workaround it I do some git checkout to use the same source code installed in the scicore cluster
   cd /usr/local/src/freebayes/
   && git checkout 39e5e4b # this is the git commit for freebayes release 1.1.0
cd /usr/local/src/freebayes/SeqLib && git checkout cce1e410ef6d2ac64972f5cacc8a0f9b86cecd8

cd /usr/local/src/freebayes/SeqLib/bwa && git checkout 0f298ce22c5c825c506129bf242348a31630c382

cd /usr/local/src/freebayes/SeqLib/fermi-lite && git checkout 5bc90f8d70e2b66184ecbd223a3be714c914365

cd /usr/local/src/freebayes/bamtools && git checkout e77a43f5097ea7e04e332e76504936b2464d49bba

cd /usr/local/src/freebayes/htslib && git checkout 0f298ce22c5c825c506129bf242348a31630c382

cd /usr/local/src/freebayes/fermi-lite && git checkout 5bc90f8d70e2b66184ecbd223a3be714c914365

cd /usr/local/src/freebayes/htslib && git checkout 0f298ce22c5c825c506129bf242348a31630c382

cd /usr/local/src/freebayes/intervaltree && git checkout dbb4c513d1ad3baac516fc1484c995da9f9b42838

cd /usr/local/src/freebayes/vcflib && git checkout 5e3ce04f758c6df16bc6d0247d25d2e6e5

cd /usr/local/src/freebayes/vcflib/fsom && git checkout a6ef3188bd347c53189384aeef7670c06ce89a3

cd /usr/local/src/freebayes/vcflib/intervaltree && git checkout 5e3ce04f758c6df16bc6d0247d25d2e6e5

cd /usr/local/src/freebayes/vcflib/smithwaterman && git checkout 84c08d6eae271d87fbcb1871da206ec2041e96

cd /usr/local/src/freebayes/vcflib/fastahack && git checkout 5e3ce04f758c6df16bc6d0247d25d2e6e5

cd /usr/local/src/freebayes/vcflib/filevercmp && git checkout 1a9b779b93d024400274794d402106907b71b7

cd /usr/local/src/freebayes/vcflib/tabixpp/htslib/ && git checkout 0f298ce22c5c825c506129bf242348a31630c382

cd /usr/local/src/freebayes/vcflib/tabixpp && git checkout 0012f86dc22b1c757b3ba381959560d92473e

cd /usr/local/src/freebayes/vcflib/multichoose && git checkout 73d35d59a185f357298e7a5247484a5

cd /usr/local/src/freebayes/vcflib/googletest/ && git checkout d25acc90bc3a8c420a9cd1f03303c1cdd7fe0

cd /usr/local/src/freebayes/
make
make install

echo "Installing vcftools-0.1.14"
cd /usr/local/src/
wget https://github.com/vcftools/vcftools/releases/download/v0.1.14/vcftools-0.1.14.tar.gz
tar xf vcftools-0.1.14.tar.gz
cd vcftools-0.1.14
./configure
make
make install

echo "Installing seqtk-1.2-r94"
cd /usr/local/src/
tar xf v1.2.tar.gz
cd /usr/local/src/seqtk-1.2/
make
cp /usr/local/src/seqtk-1.2-seqtk /usr/local/bin/

echo "Installing SRA-Toolkit/2.8.1-3-centos_linux64"
cd /usr/local/src/
tar xf sratoolkit.2.8.1-3-centos_linux64.tar.gz

cp -ra /usr/local/src/sratoolkit.2.8.1-3-centos_linux64/bin/* /usr/local/bin/
rm -fr /usr/local/src/sratoolkit.2.8.1-3-centos_linux64/

yum -y clean all
B. SLURM submission script for bioinformatics pipeline2

#!/bin/bash

#SBATCH --job-name=pipeline2-containers
#SBATCH --partition=shi
#SBATCH --time=02:00:00
#SBATCH --cpus-per-task=4
#SBATCH --mem=10G
#SBATCH --output=containers.log

module load Singularity/2.3

# define the number of cpus to use
# you can use env vars provided by your scheduler or hardcode the number of cpus to use
export NUM_CPUS=$(SLURM_CPUS_PER_TASK)

export SINGULARITY_CONTAINER="/pipeline2.img"

#############################################################
# YOU SHOULD NOT NEED TO CHANGE ANYTHING BELOW THIS LINE
#############################################################

# create output directories if they don’t exist
[ -d output/bowtie2 ] || mkdir -p output/bowtie2
[ -d output/samtools ] || mkdir -p output/samtools
[ -d output/freebayes ] || mkdir -p output/freebayes

RefGenome="/scicore-pipeline2-input-data/RefGenome/igenomes/Saccharomyces_cerevisiae/Ensembl/R64-1-1/Sequence/"

echo 'aligning reads with bowtie2...

singularity exec $SINGULARITY_CONTAINER bowtie2 -p ${NUM_CPUS} -x $RefGenome/Bowtie2Index/genome -q -1 scicore-pipeline2-input-data/fastq/SRR5511076_1.ds.fastq -2 scicore-pipeline2-input-data/fastq/SRR5511076_2.ds.fastq -S output/bowtie2/yeast_reseq_ds-bt2aln.sam

echo 'SAM to sorted BAM...

singularity exec $SINGULARITY_CONTAINER samtools view -bS output/bowtie2/yeast_reseq_ds-bt2aln.sam | singularity exec $SINGULARITY_CONTAINER samtools sort - -threads ${NUM_CPUS} -o output/samtools/yeast_reseq_ds-bt2aln.s.bam

#rm output/bowtie2/yeast_reseq_ds-bt2aln.sam

echo 'calling variants with FreeBayes...

singularity exec $SINGULARITY_CONTAINER freebayes -f $RefGenome/WholeGenomeFasta/genome.fa output/samtools/yeast_reseq_ds-bt2aln.s.bam > output/freebayes/yeast_reseq_ds-bt2aln.s.vcf
C. Full runner script example for bioinformatics “pipeline2”

```bash
name: sciCORE Pipeline 2
version: 0.1

build:
  type: bootstrap
  source: Pipeline2
  size: 2048

substitutions:
  RefGenome:
  "./scicore-pipeline2-input-data/RefGenome/igenomes/Saccharomyces_cerevisiae/Ensembl/R64-1-1/Sequence/

run:
  commands:
    - "[ -d output/bowtie2 ] || mkdir -p output/bowtie2"
    - "[ -d output/samtools ] || mkdir -p output/samtools"
    - "[ -d output/freebayes ] || mkdir -p output/freebayes"
    - "echo 'aligning reads with bowtie2...''"
    - "singularity exec {image} bowtie2 -p ${{NUM_CPUS}} -x {RefGenome}/Bowtie2Index/genome -q -1
      scicore-pipeline2-input-data/fastq/SRR5511076_1.ds.fastq -2
      scicore-pipeline2-input-data/fastq/SRR5511076_2.ds.fastq -S
      output/bowtie2/yeast_reseq_ds-bt2aln.sam"
    - "echo 'SAM to sorted BAM...''"
    - "singularity exec {image} samtools view -bS output/bowtie2/yeast_reseq_ds-bt2aln.sam |
      singularity exec {image} samtools sort --threads ${{NUM_CPUS}} -o
      output/samtools/yeast_reseq_ds-bt2aln.s.bam"
    - "echo 'calling variants with FreeBayes...''"
    - "singularity exec {image} freebayes -f {RefGenome}/WholeGenomeFasta/genome.fa
      output/samtools/yeast_reseq_ds-bt2aln.s.bam" > output/freebayes/yeast_reseq_ds-bt2aln.s.vcf"

test:
  test_files:
    - scicore-pipeline2-input-data.tar.gz
    - scicore-pipeline2-input-data
    - output.md5
  prepare_commands:
    - "echo '33921b7c4b3d3f3da638d9aa517c0a07  -' > output.md5"
    - "wget -O scicore-pipeline2-input-data.tar.bz2
      http://users.scicore.unibas.ch/~escobar/data/scicore-pipeline2-input-data.tar.bz2"
    - "tar xzf scicore-pipeline2-input-data.tar.gz"
  validate_commands:
    - "singularity exec {image} vcftools --vcf output/freebayes/yeast_reseq_ds-bt2aln.s.vcf
      --FILTER-summary --stdout | md5sum -c output.md5"
```